連想記憶を用いたデータフローマシンの一構成法

A Data Flow Machine Architecture Based on Associative Memory

長谷川 隆三　三上 博英　雨宮 真人
Ryuzo Hasegawa　Hirohide Mikami　Makoto Amamiya
日本電信電話公社　武蔵野電気通信研究所
Musashino Electrical Communication Laboratory, N.T.T.

1. まえがき

ハードウェア技術，素子技術の飛躍的な発展に伴い、ソフトウェア生産性の問題が大きくクローズアップされ、記述，検証，保守の容易なプログラミングシステムの確立が強く要求されている。
この要求に対し、プログラミング言語，方法論の立場から構造化プログラミング，データ抽象化法等いくつかの進展が見られるものの、ソフトウェア問題に決定的な解を与えているとは言い難い。これに対し、従来のノイマン形計算機を基本とした逐次的プログラムの世界から脱却して、関数概念に基づく非逐次的プログラムによって処理を記述し、プログラムの記述性，検証容易性を得ようとする関数形言語，非手続形言語[1],[2]の重要性が認識されてきている。

一方、LSI 技術の進歩は、計算機アーキテクチャに対し、並列処理，分散処理による高速処理の実現可能性を示唆しており、与えられた並列処理性をうまく反映させる計算機方式の確立は、今や重要な課題となっている。

データフローの概念はこれら２つの問題，即ち関数形言語によってプログラミングされた問題の持つ並列処理性をうまく引出し、それを効率的に実行すること，を解決する有効な計算概念であると考えられる。

データフローの概念では、処理本体に必要とされるデータ（パラメータ）が揃えば、その本体の実行が起動される。したがって各処理本体間の実行順序関係はデータの授受で規定されることになり、並列処理（Concurrency）のひとつの有効な概念化である。

処理本体のレベルをどこに設定するかで、種々の見方が生じてくる。[3] 既存のオペレーティングシステムにおけるタスクヌはプロセスの起動契機はイベント或いはメッセージであり、これもデータ駆動と見なせる。Hoar，Brinch Hansen によって提案されている Communicating Sequential Process[4],[5] の概念も、プロセスを処理本体として位置づけると、プロセス間交信をデータフローと見ることができる。

手続き或いは関数を処理本体として位置づけ、その呼出し（パラメータの引渡し）をデータフローとして捉えることもできる。この概念からの研究には、DDM1[6] や Keller らの Lisp マシン[7] の試みがある。

Dennis[8]，Arvind[9]，Treleaven[10] 等は処理本体を ALU 等ハードウェアで実現する演算器レベルで捉えている。前２者の概念のアーキテクチャでは処理本体が従来形の逐次実行形プロセッサで実現できるのに対し、演算器を処理本体として考えるアーキテクチャでは計算機構造そのものがノイマン形と大きく異なってくる。

処理本体を演算器レベルで捉えることにより、低レベルでの並列処理性が引出せ、またハードウェアの基本演算から関数性が実現でき、関数性の徹底が可能となる。しかし Dennis 等のデータフロープロセッサでは、loop 構造の導入や関数の階層性がアーキテクチャ構造に反映されてない等いくつかの不十分な点がある。

我々のデータフローマシンは関数形言語によ

るプログラムを効率的に実行することを目的としている。特にloop概念を排除し、loop構造をtail recursionに置き換えて実行することを前提としている。また関数形言語により記述されたプログラムの持つ並列処理性を有効に引出すため、並列実行のレベルをプリミティブな演算器レベル、および定義された関数の起動レベルの2つで捉えている。第1のレベルはアーキテクチャ上データ駆動により実行制御を行うComputing Moduleによって実現される。原理的にはDennis等のものと同じであるが、実行可能命令の同時取出しを有効に行うため、命令語用のメモリとして連想メモリを用いたこと、loop制御の代りにtail recursionの効率的制御を実現したことが大きく異なる。

第2のレベルの並列性は関数対応に割付けられたComputing Moduleの並列実行によって得られる。Computing Module間の交信は関数のCall/Returnの時にのみ生ずる。

本稿では関数性の観点から従来のデータフローモデルについて議論し、loop構造を排除したデータフローモデルの提案を行う。次に、このモデルの特質を最大限に生かし、並列処理性を実現させるのに連想メモリが適していることを述べ、連想メモリをベースとしたComputing Moduleの構成について、そのインストラクション・メモリの構成、マシン構造を中心に議論する。

2. データフローモデル

本節では、データフローの概念を基礎として提案された計算モデル(ここではデータフローモデルと呼ぶ)を考察し、それが備えるべき重要な性質およびモデル上の問題点について言及する。主としてDennis[11], Kosinski[12]等のモデルを念頭に置いているが、その他のモデルについてもほぼ同様のことが指摘できる。

2.1 データフローモデルと関数性

従来の計算モデルではStorageの概念が陽に存在し、globalなstate(Storageの内容)に種々の方法でアクセスすることにより処理が進められるが、この原理、性質は、既にフォン・ノイマン・ボトルネックとして知られる様々な問題を引起こす。[1]

並列処理、プログラミング方法論の観点からglobal stateの除去が肝要であり、local stateのみに基づく計算モデルの導入が必要である。データフローモデルは、このような要求を満たしうるモデルであり、その基本原理は、

1. data driven(オペレーションの実行順序は逐次的ではなく、オペレーションに必要なオペランドが全て揃ったか否かによって決まる。)
2. functionality(オペレーションは関数であり、副作用がない。即ち、結果は入力のみに依存し、他の計算や過去の履歴には影響を受けない。)

の2点に集約される。

データフローモデルの関数性は、並列処理に必要なlocalityの要求を満たし、それから引出される実行順序の非決定性はparallelismの自然な追求を可能とするため、この性質は並列計算モデルの重要なファクタである。

一方、これ迄に提案されたデータフローモデルについて、Dennisのモデルを中心に問題点を整理すると次のようなことが言える。

1) デッドロック、トークンの衝突の問題がある。これはプログラム構造上の問題とも絡むが、図1に示すように、発火可能であっても発火できないノードや、決して発火することのないノードが存在したり、演算の非同期性のためにデータパス上でトークンの衝突が起こる場合が生ずる。
2) 他のノードとは発火規則を異にするiteration制御用のMergeノードが存在する(図2参照)。このノードの発火規則は先に述べたfunctionalityの原則からはずれる。

図1 deadlock, conflict

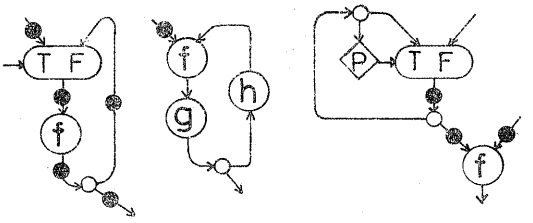
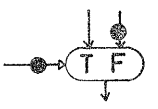


図2 発火可能なMergeノード



3) オペランドが揃っていても、出力アーク上にトークンがないことを確かめなければ発火できないという規則がある。即ち、トークンの衝突回避のための行先チェック機構や、データパス上においてバッファリング機構を必要とする。

以上述べた問題はすべて、

i) loop構造をモデル上に反映させたこと

ii) アーク上を2度以上トークンが流れることを許したこと

に起因している。したがって、このような問題を根本的に解決するには、i), ii)の要因を排除したモデルの導入が必要である。端的にこれを行うには、図3に示すように、関数性を損なうiterationはそれと等価なtail recursionに置き換えればよいわけであるが、実際上は幾つかのトレードオフ点が存在する。

iterationは、前の演算結果が得られない限り次の計算に進めない、逐次的性質の処理である。したがって、iterationの制御をうまく行ったとしても並列性が本質的に向上することはない。但し、iterationは資源を繰返し使用するため使用効率は高く、階乗のように単純な計算の場合、一般にrecursionよりも低コストでしかも高速に処理することが可能である。

一方、recursionはhistory sensitivityを持たず、プログラムの検証，解析や並列処理を行う上で有用と思われる関数性など多くの利点を提供するが、関数コピーを必要とし、実現上コスト高を招く。

以上の点がrecursionをとるかiterationをとるかの主要なトレードオフ点である。

iterationを認める立場では、new constructの導入によりiterationを関数的にみせる方法が提案されているが、[9],[13] 我々は次の理由により、iterationをモデルの上で排除し、これをすべてrecursionに置き換える方法を採用した。

1) データフローモデルにおいて関数性は極めて重要である。

2) tail recursionは論理的には関数コピーを必要とするが、3.2節で述べるように実現上は必ずしもコピーをしなくてよい。

3) new constructは、loop中のassignmentに対し、new valueやold valueの特殊な取扱いを必要とするので、マシンの論理構造の簡明さを失っている。

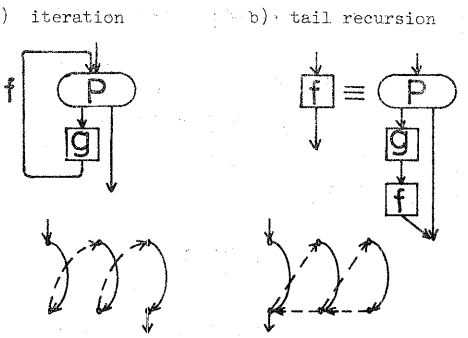


図3 繰返しと等価な再帰表現

4) モデルから高級言語に至るまで、構造的な差異が少ないことが望まれる。

2.2 修正データフローモデル

2.1の考察結果を踏まえた新モデルについて述べる。このモデルは、

i) iterationはrecursionに置き換える。

ii) アーク上を2度以上トークンを流さない。即ち単一代入規則を徹底する

の2点を骨子として従来モデルを修正したものである。ここでは、従来モデルと差異のない部分は省略し、特徴的な部分のみを示す。

• データフローグラフは、ノード，アークおよび関数定義よりなる。

• 入力アーク上に必要なトークンがすべて揃った時点で、ノードが発火可能となり、1回だけ実行する。アーク上を2度以上トークンが流れることを禁止する。論理的には、実行の終了したデータフローグラフは消滅する。単一代入規則の遵守によりトークンの衝突がなくなるので、出力アーク上にトークンがないことを確かめる必要はない。

• 関数定義は、関数名，関数定義箱，グラフ表現された関数本体，およびパラメータ受渡しのための仮アークよりなる。関数の定義は、関数定義箱の中に関数本体を書き、それに関数名を付与することで行われる。関数呼出し(Call)ノードが実行されると、必要に応じて関数本体がコピーされ、仮アークとCallノードに入る実アークとの間でパスがはられる。

• 同じ関数名を持つものを2度以上定義しては

ならない。同じ関数を繰返し使用する場合は、その都度新しいコピーが作られ、それに対してユニークな関数名が与えられる。

• トークンの大きさ，種類は任意である。

図4に、本モデルで扱うプリミティブ・ノードのタイプを示す。

2.3 モデル実現上の問題点

データフローグラフを実行する方法としては、

i) 各ノードにそれぞれ専用の演算器を対応させ、ノード間の結合関係をハード的結合関係に対応させる（即ち、データパスをはりめぐらす）方法，と

ii) データフローグラフを機械語で表現して、これをメモリに格納し、読出した機械語の命令コードに従って演算器を割当てる方法，

が考えられるが、物理的資源が有限である点、および実現性の観点から、ii)の方法を採用する。以下、これを基礎に議論を進める。

2.2で述べたデータフローモデルの特長を十分に生かしてマシンを実現するには、先ず次の問題を解決しなければならない。

1) 実行可能な命令を複数個同時に取出し得ること（同時発火）
2) リンクノードを実現する際、結果の分配先すべてに同時にトークンを分配し得ること（同時分配）
3) 複数のノードで結果のトークンが生じた場合、それぞれ並行して、これらを必要とするノードへ分配し得ること（並列分配）
4) コントロール・ノードは高速に実行し得ること

以上の問題は、インストラクション・メモリの実現の仕方に大きく依存する。これ迄に提案されたデータフローマシンは、十分にこの点を解決しているとは言い難い。例えば、LAU[13]やRumbaugh[14]のデータフローマシンのように、インストラクション・メモリとして従来のメモリを使用する方式は、インタリーブなどの技術によりメモリアクセスの並列性を高めることができるが、結果の分配，命令の取出しにはポインタを辿る操作が必要であるので、逐次的な処理の部分が残る。これに対し、Dennisのマシンでは命令セルは全てレジスタ構成であるため、並列読出し・並列書込みの問題は解決されるが、依然としてアドレス概念を持込んでおり、メモリ管理が複雑になる点や容量，実現性などの点で問題がある。

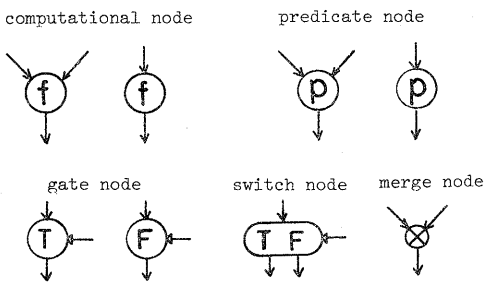


図4 プリミティブ・ノード

以上述べたデータフローモデル実現上の一般的な問題の他に、本モデルの実現に際してはさらに、発火後のグラフの消滅，関数コピーの際の空き領域の管理，iterationをrecursionに展開する場合のコピーの無駄の除去，等の問題を解決する必要がある。

3. 連想記憶によるデータフロー機械語

2.3で論じたモデル実現上の要求を満たす方式として、インストラクション・メモリを全面的に連想記憶で実現する方式を採った。その理由は次の点にある。

① タグ，ラベルによる連想アクセスにより、実行可能命令の同時読出し，オペランドの同時書込みが可能となり、機械語の中にアドレス概念を持込む必要性がなくなる。

② 実行後の命令セルの無効化，関数本体のコピーの際の空き領域の管理が、連想アクセスにより単純化される。

一方、連想記憶を用いたことによる実現上の問題点としては、

① 同時に複数個の実行可能命令を見つけることができてもその取出しが逐次的になる。

② あるラベル検索中には他のラベルによる検索が不可能になる。

等があり、これらの解決にはアーキテクチャ上工夫を要する。

上記問題を解決するアーキテクチャ構成については次章で述べることにし、まず本章では、データフロープログラムを連想記憶内で表現す

る機械語の構成について説明し、関数リンケージの制御法について述べる。尚、議論の前提として、ユーザの書いた関数形言語によるプログラムはコンパイラによって本機械語に翻訳されているものとする。

3.1 機械語構成

データフロー機械語のフィールド構成を図5に示す。ノードのタイプはモデル上5種に類別されるが、機械語フィールド構成上からは2項演算/比較、単項演算/比較、ゲート制御、スイッチの4タイプに分けられる。各タイプの機械語は name field, operation code field, operand field よりなる。また各機械語には、3種の tag field が付けられている。以下、各 field の説明を行う。

(a) name field : name field は func # part (name.func # と記す) と actor name part (name.actor) よりなる。actor name はノードに付されたラベルに対応し、コンパイル時に決定付与される。func # は関数の activation 毎にその名前が区別して付与される。name field の LSB はスイッチ命令に対して意味を持つ。この命令の第2オペランドの値 (true/false) によって 1/0 にセットされる。即ち、スイッチ命令の名前は初め name.actor * と置かれており、第2オペランドの到着時にその値により name.actor 0 或いは name.actor 1 と決定される。

(b) operand field : 2オペランドの命令の場合、2つの operand field opr1, opr2 があり、1オペランド命令の場合は opr1 のみである。各 operand field は name part (opr.name) 及び value part (opr.value) からなる。命令 P_1 の演算結果を命令 P_2 の第1/2オペランドとする場合、P_2 の opr1.name / opr2.name には P_1 の name.actor が記されている。実行制御においては、P_1 の name.actor が記されている opr.name を持つ命令 (P_2) を連想アクセスにより認識し、その opr.value に P_1 の演算結果を書込む。この時は同じ opr.name を持つ全ての命令語の opr.value にも同時に値が書込まれることになる。

連想アクセスにおいて name.actor と opr.name のマッチングは同一の name.func # を持つ命令語についてのみとられ、よって activation による区別がなされる。opr.name の LSB はこれがスイッチの結果を指定する場合にのみ意味を持つ。即ちスイッチの T/F 側 (name.actor 1 / name.actor 0) を指定するとき LSB は 1/0 と置かれる。

(c) operation code field : 演算コード。デコードされ対応する演算ユニットが選ばれる。

(d) tag field : enable tag (t_0), operand arrived tag (t_1, t_2) よりなる。各 t_i ($i = 0, 1, 2$) には 3.2 節で述べる tail recursion 制御のためのリセット情報用ビット t_i^R が付随している。t_i^R の値は、コンパイル時に以下のように定められる。

$t_0^R \leftarrow$ *if* tail recursion *then* 1 *else* 0

$t_i^R \leftarrow$ *if* opr i = constant *then* 1 *else* 0

t_0 はその命令語セルが有効か無効かを示す。プログラムは全て関数として定義されており、それが呼ばれたときその関数全体がコピーされる。コピー時、$t_0 = 0$ である命令語を連想アクセスし、そこに各命令を書込んでいく。$t_0 = 1$ のときはこの命令が活性化されオペランドの受付け可能であることを示す。$t_1, t_2 = 1$ のときは既に opr.value に値が書かれていることを示す。$t_0 \wedge t_1 \wedge t_2 = 1$ のときは実行可能を示し、この条件を満たす命令が実行される。命令が実行されると全ての tag はリセットされる。t_0 は関数本体のコピーが終了した時点で、その全ての命令に対してセットされる。t_1, t_2 は opr1.value, opr2.value への値の書込みが終了した時点でセットされる。

図6に、本機械語のコーディング例を示す。

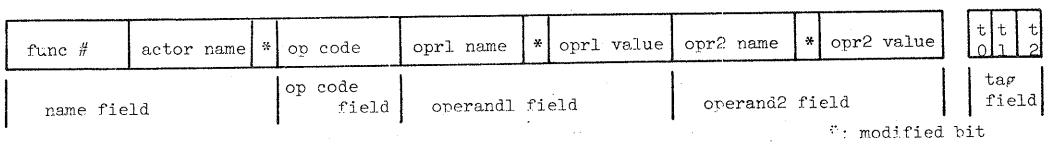


図5　データフロー機械語のフィールド構成

3.2 関数リンケージ

図7に示すように、モデルで示された関数リンケージは、Call命令，Link命令，Return命令，及びGate命令を用いて実現される。図で a_1 ～ a_n，b は実アーク，x_1 ～ x_n，y は仮アークである。 Gate命令は実アーク上に全てのトークンが揃ったとき Call命令を起動させる。Call命令は opr2. value で示された関数 f の本体のコピーを作り、そのときその関数に付された new func# を値として持つ。 Link命令は opr1. value に new func# が与えられると実行され、opr2. value に置かれている実アーギュメントを関数本体に送り出す。アーギュメントの引渡しは、new func# を name. func# に持つ全ての命令のうち、x_1 ～ x_n を opr.name として持つものに対してなされる。

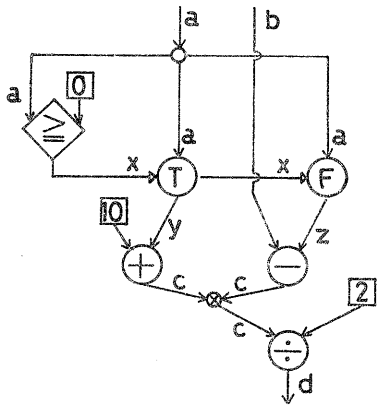
関数本体からのリターンは次のように実行される。呼び側には関数値引渡しを制御するために、name. actor を y，y' とする Link1，Link2 命令が置かれる。また、関数本体中には、Return 命令が置かれる。

Return 命令の name. actor は、Link2 命令から opr2. value に書かれている値 b が送られてくる返来定である。呼び側の Link1命令は呼び側の func# を本体側の Return 命令に引渡す。関数本体の計算結果が Return 命令の opr2. value に書込まれると Return 命令が実行可能となる。 Return 命令の実行では次の処理が行われる。

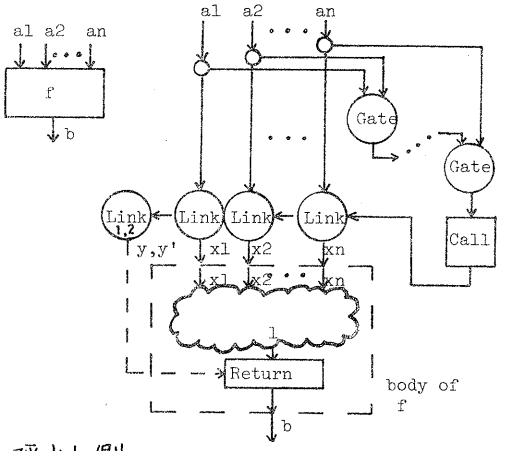
① opr2. value の値を呼び側に引渡す。引渡

図6 コーディング例

```
c← if a≧0 then 10+a
                else b-a;
d← c÷2;
```



f#	a.n.	op.c	opln	oplv	op2n	op2v	t0	t1	t2
	a						10		
	b						10		
	x	≧	a			0	10	00	10
	v	T	a		x		10	00	00
	z	F	a		x		10	00	00
	c	+		10	y		10	10	00
	c	−	b		z		10	00	00
	d	÷	c			2	10	00	10



呼出し側

f#	a.n.	op.code	operand1 name	operand1 value	operand2 name	operand2 value
100	s1	Gate	a1		a2	
100	s2	Gate	s1		a3	
100	sn-1	Gate	sn-2		an	
100	c	Call	sn-1			f
100	x1	Link	c	(newf#)	a1	
100	x2	Link	c	(newf#)	a2	
100	xn	Link	c	(newf#)	an	
100	y	Link1	c	(newf#)		(oldf#)
100	y'	Link2	c	(newf#)		b

関数本体

200			x1			
200					x2	
200			xn			
200	y'	Return	y		1	(r.v.)

図7 関数リンケージ

しの際は、opr 1. value にある old func# を name. func# とする命令のうち、opr. name が b であるものとマッチングがとられる。
② 関数本体の全命令を無効とする。即ち t_0 を全て0とする。

次に tail recursion の実行制御について述べる。従来の iteration 構造のプログラムは図8(a)に示すような tail recursion で記述できる。原則に従えば、関数 f の本体はそれが呼ばれる毎にコピーが作り出されることになる。しかし、tail recursion の性質から f が再度呼ばれるときには、呼び側の f の環境情報は不要なものとなっている。したがって概念上図8(a)のような再帰構造は、実行制御上図8(b)のように変換しても論理的矛盾は生じない。即ち、関数 f の結果が全て揃った時点で、これらの値を再び関数 f のアーギュメントとして与え、関数 f を実行させてやることにより、関数のリンケージ及び関数本体のコピーに要するオーバヘッドをなくすことができる。

tail recursive な関数本体の各命令は、それが複数回実行されるため、命令の tag t_i ($i=0,1,2$) は実行後0にリセットされるのではなく、t_i^R の値にリセットされる。
そして最後に Return 命令に到達し recursion block から出るときに初めて、t_0 が0にリセットされる。

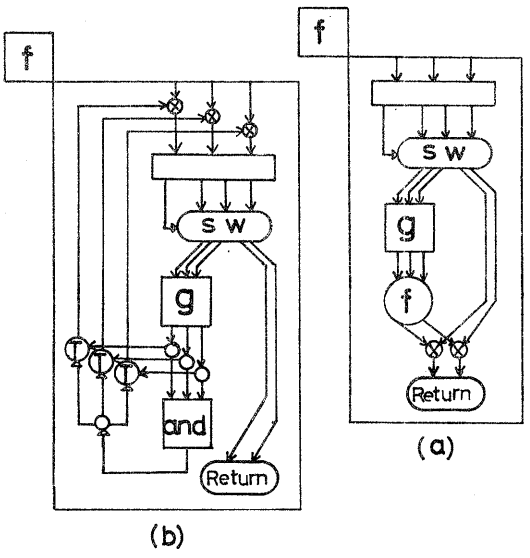


図8 tail recursion の実行制御

4. データフローマシン構成

4.1 データフローマシンシステム

我々のデータフローマシンシステムは、データ駆動で動作する多数個のデータフロープロセッサ（これは Computing Module, CM と呼ばれる）を結合系（Connection Network, CN と呼ばれる）を通じて結合させたものとして構成される。（図9参照）

本システム上でのプログラムは関数形言語で行われることを前提として考えているが、各定義された関数の本体は各 CM に割付けられる。したがって CM 間の交信が引き起こされるのは関数呼出しが生じたときのみである。

関数形言語では徹底した構造化プログラミングを余儀なくされ、よって各関数間には木状の階層関係が成り立つ。しかもこの関係はコンパイル段階で抽出でき、CM 間の交信が局所的に行われるように、静的な関数割付けを計画することが可能である。　CN の構成には自由度の高い通信路設定が要求されるが、通信路の設定方法には、関数を CM に割付けたとき同時にパスを設定する半固定方式，関数の呼出しが生じたときに動的にパスを設定するオンデマンド方式等いくつか考えられる。現在、関数呼出しの頻度，1回の呼出しで転送されるデータ量の点から、これらの方式の検討を進めているが、詳細は別途議論することとする。本稿では以下、特にデータフローマシンシステムの基礎となる CM の構成について論ずる。

4.2 CM（Computing Module）構成

CM は単体としても動作しうるデータフロー

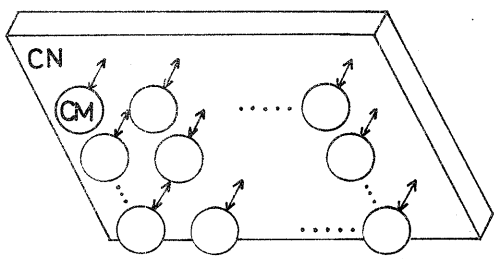
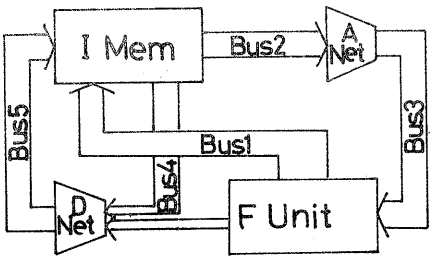


図9 データフローマシンシステム



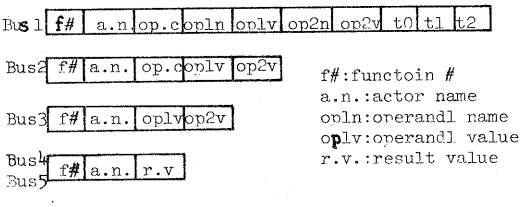


図10　CM（Computing Module）構成

プロセッサである。全体構成を図10に示す。

CMは、データフロー機械語を保持するインストラクションメモリ（I.Mem），複数の演算ユニット（F.Unit），調整／分配ネットワーク（A.Net／D.Net）およびこれらを結合するバスから構成される。以下、簡単にCMの動作を述べる。

実行可能になった命令は、命令パケットとして組立てられ、Bus2を経由してA.Netに送出される。A.Netでは命令コードをデコードし、Bus3を経由してそれぞれ専用の演算ユニットへ命令パケットを送る。演算ユニットでの実行が終了すると、結果の値と送り先情報を組にした結果のパケットが、Bus4を経由してD.Netへ送られる。D.Netは結果のパケットを受取ると、それを待つ命令が格納されているメモリ・エリアを選別し、Bus5を介して当該エリアにパケットを送出する。

T/F GateやLink命令のように、データの流れのみを制御する命令の場合は、演算ユニットへ送らず直接インストラクションメモリ内で結果のパケットを作り、Bus4を介してD.Netへ送出する。

命令パケットがCall命令であれば、Call演算ユニットは2次メモリから関数本体を読出し、それにfunc#を付与して、Bus1を介しインストラクションメモリへロードする。

Bus1～Bus5上のパケット形式については図10を見られたい。

4.3 インストラクションメモリ構成

インストラクションメモリは、図11に示すように、書込み（Bus1, Bus5）用コントローラ，読出し（Bus2，Bus4）用コントローラおよび連想メモリからなる。

連想メモリは、機械語の1語を収容するセルを単位として構成される。

連想メモリ全体は、独立に書込み動作が可能なセグメントおよび独立に命令取出し動作が可能なブロックに分割されている。

セグメントは、ひとつの結果のパケットに対してラベルによる検索を行い、該当する複数個のセルへ同時に値を分配する単位であり、また関数コピーをインストラクションメモリへロードする場合の格納単位でもある。

このように連想メモリを複数のセグメント，ブロックに分割して構成することにより、3章の冒頭で述べた、結果のパケットの分配，実行可能命令の取出しが逐次的になるという問題を解決している。しかし、本方式を有効利用するには、プログラム配置の問題を解決する必要がある。即ち、ある結果を待つ複数の命令に対し同時分配するときは、それらの命令をセグメント内に局在化しなければ、検索がメモリ全体

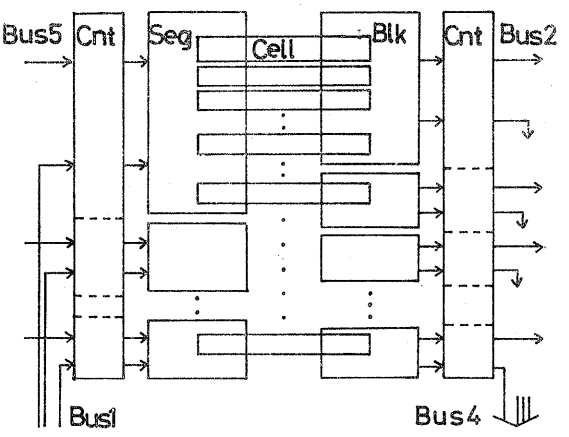


図11　インストラクションメモリ構成

に及ぶのでメモリの分割効果はなくなる。これに対し、独立な演算結果を並列分配するときはこの逆のことが言える。したがって、関連のあるまとまった処理はセグメント内に局在化させ、関連のない処理は別々のセグメントに置くことが望ましい。　上記の問題や，メモリ分割の最適サイズ,詳細構成については別途検討する。

4.4　調整・分配ネットワーク

A.Netの構成を図12に示す。A.Netは複数個のアービタ（Ar）およびデコーダ（D）よりなる。　デコーダは入力パケットの命令コード部をデコードして出力線を選択し、その上に命令コード部を取除いたパケットを送出する。
Bus2側のArとBus3側のArには本質的差異はなく、取扱うパケットが異なるだけである。

このように複数個のデコーダを介することにより、パケットを複数本のバスに整列させることが可能となる。

D.Netは図13に示すように、A.Netと同様の構成をとる。A.Netとの違いは主にデコーダの機能である。D.Netのデコーダは送られたパケットの行先ラベルから、結果を待つ命令のSegment#を決定し、それに対応する出力線上にパケットを送出する。ここでは、Segment#を決定する方法の詳細については省略するが、デコーダ部にfunc#とSegment#の対応表を持たせる方法や、機械語の中に陽にSegment情報を入れておく方法などが考えられる。

4.5　演算ユニット

演算ユニットの構成を図14に示す。演算ユニットはアロケータ（Al）および各演算対応に専用化した演算器からなる。　Alには同一タイプの演算器が複数個つながれる。Alネックが生じるときは、Alを複数個にして負荷のバランスをとる。このためには、A.NetのBus3側の構成を少し変えるだけでよい。

Alは命令パケットを受取ると、空き状態の演算器を見つけ、それにパケットを送出する。空き演算器がなければ、演算器が空きになるまでAlは待たされる。

図14において、Co, Ciは他のCMと通信を行う演算器であり、結合系CNにつながれる。

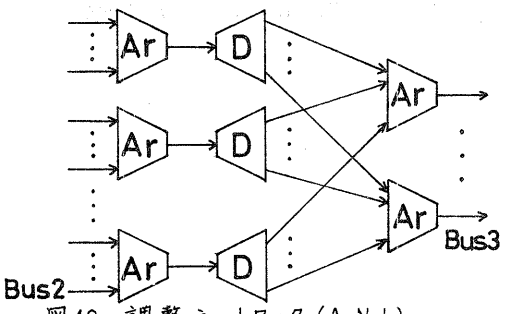


図12　調整ネットワーク（A.Net）

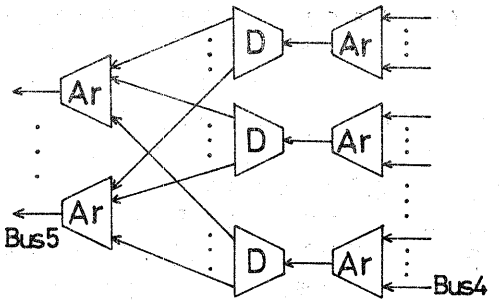


図13　分配ネットワーク（D.Net）

5.　あとがき

関数形言語で記述されたプログラムを効率的に実行するマシンアーキテクチャとしてデータフロープロセッサの構成法について議論した。

先ずDennis等の従来のデータフローモデルがloop構造を許していることに起因する問題点を考察し、loop構造を排除して、ノードは高々1度しか発火しないという徹底した関数性を追求するデータフローモデルを提案した。

次にこのモデル上の関数性をうまく生かし、並列実行性を実現するために、連想メモリを用

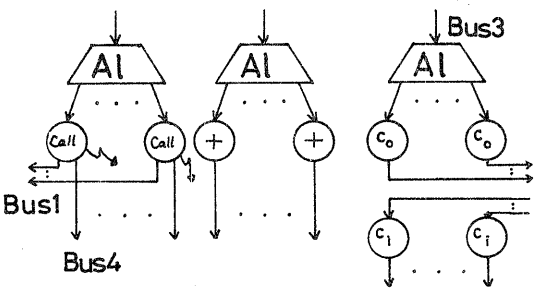


図14　演算ユニット（F.Unit）

いたインストラクションメモリの構成法及びそのマシンアーキテクチャを提案した。連想メモリを用いる理由は、現在高価なものであるが、将来LSI化によりその価格は問題にならなくなるだろうとの予測によるものである。特に本方式による連想アクセスは単純であり、そのLSI化は現実に可能であると思われる。

本方式での問題点として、関数が呼ばれる度に行われる、関数本体のコピーに要するオーバヘッドがある。しかしそのコピーは単なるロード処理であり、また関数の並列実行の陰にそのオーバヘッドが隠せると思われる。またArvindら[9]のcolored tokenの場合、そのcolorの制御のオーバヘッドを考えると、コピーすることの利点の方が大きいと思われるが、これらの定量的な評価は今後の検討課題である。

iterationはノイマン形マシンでの資源の有効利用と高速処理の点で非常に有効な計算技法である。しかしこのiterationはプログラムを不透明にする大きな要因となっている。本稿ではiterationがtail recursionに置き換わることに、しかもtail recursionでは同じ関数本体を繰返し使用しても論理上矛盾が生じないことに着目し、再帰呼出しに際していちいち関数本体をコピーしなくて済む効率的実行制御法を提案した。

連想メモリを用いた場合の処理効率、関数コピーのオーバヘッド、またComputing Module間の交信頻度と並列処理性など、本方式の定量的評価は今後に残された問題である。

最後に日頃我々の研究に対し叱咤激励、有益な助言を与えて下さる山下統一第一研究室長に謝意を表する。

参考文献

1. J.Backus : Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs, CACM, 21, 8, 1978, PP. 613-641
2. E.A. Ashcroft and W.W. Wadge : Lucid, a Nonprocedural Language with Iteration, CACM, 20.7. 1977, PP. 519-526
3. P. Denning : Operating Systems Principles for Data Flow Networks, IE^3 Computer, July, 1978, PP. 86-96
4. C.A.R. Hoar : Communicating Sequential Processes, CACM, 21, 8, PP. 666-677
5. P. Brinch Hansen : Distributed Processes : A Concurrent Programming Concept, CACM, 21, 8, 1978, PP. 934-941
6. A.L. Davis : A Data Flow Evaluation System Based on the Concept of Recursive Locality, Proc. NCC'79, 1979, PP. 1079-1086
7. R.M. Keller, G. Lindstrom and S. Patil : A Loosely-Coupled Applicative Multiprocessing System, proc. NCC'79, 1979, PP. 613-622
8. J.B. Dennis, C.K. Leung and D.P. Misunas : A Highly Parallel Processor Using a Data Flow Machine Language, MIT CSG Memo, 134-1, 1977, P. 33
9. Arvind, K.P. Gostelow and W. Plouffe, An Asynchronous Programming Language and Computing Machine, UCI memo, 1978
10. P.C. Treleaven : Exploiting Program Concurrency in Computing Systems, IE^3 Computer, Jan. 1979, PP. 42-50
11. J.B. Dennis, J.B. Fossen and J.P. Linderman : Data Flow Schemas, In International Symposium on Theoretical Programming, Springer-Verlag, New York, 1974
12. P.R. Kosinski : A Data Flow Programming Language, Report RC 4264, IBM, Mar. 1973
13. L. Plas, et al : LAU System Architecture : A Parallel Data Driven Processor Based on Single Assignment, Proc. International Conference on Parallel Processing, 1976, PP. 293-302
14. J.E. Rumbaugh : A Parallel Asynchronous Computer Architecture for Data Flow Programs, MIT, MAC TR-150, 1975